As will be discussed below, the Office has failed to establish a *prima facie* case of obviousness. To establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by the prior art as indicated by M.P.E.P. Section 2143.03. The cited and applied references do not teach, on the one hand, the concept of using a variable of an object as a pointer to a lock, and, on the other hand, the concept of returning an unused lock to the pool of locks without having to destroy an object previously associated with the lock as recited in Claims 1, 11, and 17. Moreover, a number of cited and applied references cannot be combined, such as Brown et al. and Lindholm et al., without destroying the operation of either reference.

Prior to discussing in detail why applicant believes that all of the claims in this application are allowable, a brief description of applicant's invention and a brief description of the teachings of the cited and applied references are provided. The following background and the discussions of the disclosed embodiments of applicant's invention and the teachings in the cited and applied references are not provided to define the scope or interpretation of any of the claims of this application. Instead, such discussions are provided to help the Office better appreciate important claim distinctions discussed thereafter.

Summary of Processes and Threads

Early operating systems allowed users to run only one program at a time. Users ran a program, waited for it to finish, and then ran another one. Modern operating systems allow users to execute (run) more than one program at a time or even multiple copies of the same program at the same time. This change highlights a subtle distinction: the difference between a program and a process. A program is a static sequence of instructions whereas a process is the dynamic invocation of a program along with the system resources required for the program to run. Thus, a process, in the simplest terms, is an executing program.

Processes include one or more threads. A thread is the basic unit used by the operating system to allocate processor time. A thread can include any part of the process code, including parts currently being executed by another thread. A processor is capable of executing only one thread at a time. However, a multitasking operating system, i.e., an operating system that allows users to run multiple programs, appears to execute multiple programs at the same time. In reality, a multitasking operating system continually alternates among programs, executing a thread from one program, then a thread from another program, etc. As each thread finishes its subtask, the processor is given another thread to execute. The extraordinary speed of the processor provides the illusion that all of the threads execute at the same time. Multitasking increases the amount of work a system accomplishes because most programs do not require that threads continuously execute. For example, periodically, a thread stops executing and waits while a slow resource completes a data transfer or while another thread is using a resource it needs. When one thread must wait, multitasking allows another thread to execute, thus taking advantage of processor cycles that would otherwise be wasted.

While the terms multitasking and multiprocessing are sometimes used interchangeably, they have different meanings. Multiprocessing requires multiple processors. If a machine has only one processor, the operating system can multitask, but not multiprocess. If a machine has multiple processors, the operating system can both multitask and multiprocess.

Summary of Synchronization

Threads often require a way to communicate with one another to coordinate their activities. While there are many complex forms of communication among threads, such as events, semaphores, timers, and mutants, the simplest form is called synchronization. Synchronization refers to the ability of one thread to voluntarily stop executing and wait until another thread performs some operation. As briefly discussed in the example above, periodically, a thread stops executing and waits while a slow resource, such as a hard disk, completes a data transfer or while another thread is using a resource, such as a shared memory buffer, it needs.

As another example, consider a program with two threads, one with higher priority than the other, both threads requiring access to the same resource at some point. On a single processor system, the higher priority thread may have full access to the resource and will not have to relinquish control to the lower priority thread because the operating system gives preference to higher priority threads. However, on a multiprocessor system, both threads can run simultaneously, each on its own processor. This situation may cause each thread to simultaneously compete for access to the same resource, thereby creating the possibility of a race condition. A race condition exists when a thread modifies a resource to an invalid state, and then another thread attempts to access that resource and use it in the invalid state. One way to coordinate these two threads is through the use of synchronization.

One conventional synchronization technique is the use of a lock object. For example, a thread-safe multithreaded program typically is designed to have a resource data structure that represents a resource to be shared among threads in the program. This resource data structure can be synchronized by referencing to one of many synchronization capabilities provided by the environment in which the program will be run. Among the synchronization capabilities is the ability to create a lock object. The resource data structure also typically has a member function that accesses the resource to be shared. Whenever the member function of the resource data structure is invoked by a thread to access the resource, the member function creates a lock object and then calls the lock function of the lock object to prevent other threads from accessing the resource. When the member function finishes, the unlock function of the lock object is called to release the resource for other threads to use.

The approach of using lock objects as described above sacrifices efficiency for safety. The creation of one lock object may not take much time, but the creation of a multitude of lock objects may degrade computing performance to the point where most of the processor time is spent creating lock objects instead of doing tasks required by programs. These created lock objects also need to be maintained. Thus, not only must computing time be allocated to create these lock objects, but also to maintain them. Moreover, created lock objects consume resources (e.g., memory) that they do not even protect. Thus, a computing system becomes absurdly inefficient because a lock object exists for every object in the computing system, thereby doubling of the number of objects (and therefore spent precious computing resources) within the computing system just to enable objects to be locked.

Summary of the Invention

Applicant's invention is directed to avoid or reduce the unbounded creation of lock objects by a recycling process. More specifically, the system of the disclosed invention includes a pool of locks. The pool of locks has a set number of lock objects. By prescribing the use of lock objects from the pool of locks, the unbounded creation of lock objects is inhibited. One or more objects exist in the system of the disclosed invention, and at least one object represents a resource, such as a piece of data or a piece of hardware, that is needed by a number of threads. Each object that represents a resource has a variable, which can act to point to a lock in the pool of locks. The mechanism that causes the variable of an object to point to a lock, when a thread needs to access a resource represented by the object, is the recyclable locking mechanism of the disclosed invention. The lock returns to the pool of locks when the thread no longer needs to access the resource. The returning lock is in essence "recycled" for the use of another object, thereby conserving computing resources dedicated to creating or maintaining lock objects. Note that the lock is returned to the pool of locks whether or not the object continues to persist in the system of the disclosed invention.

<u>Summary of Lindholm et al.</u>

As spelled out by the Field of the Invention, Lindholm et al. describes his invention as follows:

> The present invention relates generally to systems and methods used in object-oriented multithreaded environments for synchronizing objects with threads of execution. In particular, it relates to an object synchronization module and associated method that <u>use a cache of monitors for synchronizing objects</u> with execution threads in a multithreaded environment.

Id. column 1, lines 5-12

More specifically, the system of Lindholm et al. is directed to caching and allocating thread synchronization constructs. To synchronize between a plurality of threads of execution and a set of objects, the system of Lindholm et al. provides an object synchronization module, which comprises a cache of synchronization constructs, a hash table containing a list of pointers pointing to allocated synchronization constructs, a list of free synchronization constructs, and a cache manager. When a requesting thread seeks synchronization with an object, the cache manager allocates a synchronization construct in the list for synchronizing the requesting thread with the object only when there is a free synchronizing construct in the list. During the allocation of the synchronization construct, a pointer in the list of pointers is caused to point to the allocated synchronization construct. Moreover, each synchronization construct contains an object identifier, which holds the address of the object being synchronized by the corresponding synchronization construct. If there are no synchronizing constructs (in other words, all of them have been allocated for synchronization among other threads and objects), the requesting thread is placed in a waiting list of the requested synchronization construct. The cache manager deallocates a synchronization and returns the synchronization construct to the free list by causing the pointer that points to the de-allocated synchronization to point to the free list.

Summary of Brown et al.

The system of Brown et al. is directed to adding synchronization capability to an object at run time via a locking mechanism, which is bound to the object by allowing the object to contain a pointer to the locking mechanism. The locking mechanism is bound to the object for the life of the object.

Unlike Lindholm et al., Brown et al. decries the use of monitors for synchronization. Brown et al. explains that a monitor is logically associated with an object. However, in the prior art, a monitor is not bound to the object. Brown et al. disparages the use of monitors as shown in the following text:

> While the use of monitors prevents race conditions from occurring, this approach can significantly degrade the performance of the information handling system. The use of monitors is a time-consuming process... Another problem with the use of monitors is that a monitor is effectively a wrapper around an operating system semaphore. The use of an operating system semaphore requires calls to the operating system, which significantly impacts the performance of the process which is executing. In addition, the monitor's structure contains information which is redundant with the operating system semaphore, such as the owning thread and recursion count. Maintaining this information in tuning two data structures is unnecessary and adds additional overhead to the system.

Id. at col. 2, line 38, to col. 3, line 34

To overcome this problem with the prior art, Brown et al. provides a locking mechanism that is bound to an object for the life of the object.

Summary of Kishimoto

The system of Kishimoto is directed to an initial program loading (IPL) system for a multi-processor system that has a plurality of processors. Each processor is connected to another processor through a communication path. The IPL system of Kishimoto prevents any one of the processors from suffering an excessive load that may be exacted by the IPL process when it is executing. Programs and data necessary for an IPL process are handled in units of function modules, which in the idiom of Kishimoto are "segments." Each processor in the system of Kishimoto requires different kinds of segments to carry out an IPL process. The segments can be combined differently depending on the function of a processor. Once requested, segments are sent to the requesting processor, which stores the segment in a segment table.

An example of a table of segments to be transmitted to a requesting processor is shown in Figure 14 of Kishimoto. In this table, the processor number of a given processor serves as the address of a space that stores segment numbers allocated to the processor. The segment numbers are represented using 32 bits. The higher 27 bits (from bit 31 to bit 5) indicate an address that stores 32 bits that show the allocation states of 32 segment numbers represented using the lower 5 bits (from bit 4 to bit 0). For example, if the lower 5 bits indicate segment numbers 00001 (1 in decimal notation) to 00011 (3 in decimal notation), the allocated segment numbers are 1 to 3 in decimal notation. In this case, the 32-bit word addressed by the higher 27 bits of the segment number that corresponds to the processor number is 000. . .001110 with lower bits 1 to 3, each being a "1" to indicate the requested segments.

The Claimed Invention Distinguished

As discussed in greater detail below, the claims of the present application are clearly patentably distinguishable over the teachings of the above-cited references. The present invention is directed to avoid or reduce the unbounded creation of lock objects by a recycling process by prescribing the use of lock objects from the pool of locks, the unbounded creation of lock objects being inhibited. It is accomplished by a system that comprises at least one thread, a pool of locks, at least one object (with a variable) that is capable of representing a resource needed by a thread, and a recyclable locking mechanism. The recyclable locking mechanism associates a lock from the pool of locks with an object using the variable of the object as a pointer when requested by the thread. The lock returns to the pool of locks without having to destroy the object when the thread no longer needs to access the resource. The returning lock is in essence "recycled" for the use of another object, thereby conserving computing resources dedicated to creating or maintaining lock objects. The lock is recycled whether or not the object, which was previously associated with the lock, continues to persist in the system of the disclosed invention.

Regarding the claims, independent Claim 1 is directed to a system. This system is recited as comprising at least one thread, a pool of locks, and at least one object. The object is recited as capable of representing a resource needed by a thread. The object is further recited as having a variable. The system yet further comprises a recyclable locking mechanism for associating a lock from the pool of locks with the object, which uses the variable of the object as a pointer when requested by the thread. Additionally, the system recites that the lock returns to the pool of locks without having to destroy the object when the thread no longer needs to access the resource.

Claims 2-9 are dependent from independent Claim 1 and are directed to further limitations of the system described above. Claim 2 is dependent on Claim 1 and recites that the recyclable locking mechanism further deassociates the lock from the object upon a second request by the thread. Claim 3 is dependent on Claim 1 and recites that the variable of the object comprises an integer. Claim 4 is dependent on Claim 1 and recites that the variable of the object comprises a set of high bits defining the pointer to a lock and a set of low bits defining a status variable. Claim 5 is dependent on Claim 4 and recites that the set of high bits comprises 27 bits and the set of low bits comprises five bits. Claim 6 is dependent on Claim 4 and recites that the set of low bits is initially set to -1. Claim 7 is dependent on Claim 4 and recites that upon the first request, the set of low bits is incremented by one. Claim 8 is dependent on Claim 7 and recites that the set of low bits after incrementation by one being greater than zero and the variable has an in-use status by a thread such that the set of high bits points to a lock. Claim 9 is dependent on Claim 7 and recites that upon the variable after incrementation by one being less than 32, the variable has a spin status such that the set of high bits is currently in the process of being set to a lock. Claim 10 is dependent on Claim 4 and recites that the recyclable locking mechanism further deassociates the lock from the object upon a second request by the thread such that the low bits are decremented by one.

Independent Claim 11 is directed to a method. This method is recited as comprising asserting an instruction by a thread to lock an object. The method further recites increasing a variable of the object. The variable, in particular, is recited as having a set of high bits for representing a pointer to a lock and a set of low bits for representing a lock status. The method yet further comprises determining whether the variable is greater than a boundary value so as to allocate the lock. The method as yet further comprises recycling the lock by returning the lock to a pool of locks when the thread no longer needs the object regardless of whether the object persists after the lock returns to the pool of locks.

Claims 12-16 are dependent from independent Claim 11 and are directed to further limitations of the method described above. Claim 12 is dependent on Claim 11 and recites an additional act for initially setting the variable of the object to minus one. Claim 13 is dependent on Claim 11 and recites that upon determining that the variable is less than the boundary value, the method waits until the variable is greater than the boundary value. Claim 14 is dependent on Claim 11 and recites that upon determining that the variable is greater than the boundary value, the method uses the set of high bits of the variable as a pointer to a lock for the object. Claim 15 is dependent on Claim 14 and recites decrementing the variable of the object and determining whether the variable is less than a minimum threshold. Claim 16 is dependent on Claim 15 and recites that upon determining that the variable is less than the minimum threshold, the method recycles the lock.

Independent Claim 17 is directed to a computer. This computer is recited as comprising a processor, a computer-readable medium, and a recyclable locking mechanism program executed by the processor from the medium to associate a lock when an object using a variable of the object as a pointer is requested by a thread. The lock in particular is recited by Claim 17 as being capable of returning to a pool of locks without having to destroy the object when the object is no longer needed by the thread. Claim 18 is dependent from Claim 17 and is directed to further limitations of the computer described above. In particular, Claim 18 recites that the variable of the object comprises a set of high bits that defines the pointer to a lock and a set of low bits that defines a status variable.

Independent Claim 19 is directed to a computer-readable medium. The computer-readable medium is recited as comprising a recyclable locking mechanism program stored thereon for execution on a computer to associate a lock with an object using a variable of the object as a pointer when requested by a thread. The lock as recited by Claim 19 is capable of returning to the pool of locks without having to destroy the object when the object is no longer needed by the thread. Claim 20 is dependent from Claim 19 and is directed to further limitations of the computer-readable medium described above. In particular, Claim 20 recites that the variable of the object comprises a set of high bits that define the pointer to a lock and a set of low bits that define a status variable.

As noted above, the Office rejected Claims 1-20 under 35 U.S.C. § 103(a) as being unpatentable in view of the teachings of the three references described above, alone much less in combination. As also noted above, applicant respectfully disagrees. The cited and applied references simply fail to teach all of the limitations of the independent claims, much less the recitations of many of the dependent claims. These claims particularly point out and distinctly claim subject matter that applicant regards as his invention and that is clearly patently distinguishable from the cited and applied references.

Focusing on Claim 1, there is no teaching or suggestion in the cited references for inhibiting the unbounded creation of lock objects, in the manner recited in Claim 1. Claim 1 succinctly defines the system that avoids or reduces the unbounded creation of lock objects by a recycling process. Claim 1 recites a recyclable locking mechanism for associating a lock from the pool of locks with one object using the variable of the object as a pointer when requested by a thread. Moreover, Claim 1 recites that the lock returns to the pool of locks without having to destroy the object when the thread no longer needs to access the resource. The remaining recitations of Claim 1 are directed to the system that allows the lock to be returned to the pool of locks to be in essence "recycled" for the use of another object, hence, conserving computing resources. The cited and applied references do not teach, on the one hand, the concept of using a variable of an object as a pointer to a lock, and, on the other hand, the concept of returning an unused lock to the pool of locks without having to destroy an object previously associated with the lock.

Unlike applicant's invention, objects in the system of Lindholm et al. lack a variable that can be used as a pointer to point to a lock object and/or be used to indicate the lock status. This minimal design avoids the complexity and the waste of computing resources associated with the use of the object synchronization module of the system of Lindholm et al. It should be noted, however, that each pointer in the list of pointers of the system of Lindholm et al. exists in the context of the object synchronization module to point to a synchronization construct, and therefore, these pointers cannot be likened to the variables of objects of applicant's invention. In fact, objects in the system of Lindholm et al. do not point to allocated synchronization constructs but instead allocated synchronization constructs point to corresponding objects. To accomplish this, the system of Lindholm et al. requires significant overhead, such as a cache of synchronization constructs, a cache manager, a hash table, a hash table address generator, a free list, a cache size controller, and a hash table size controller, in order to manage synchronization constructs so that they can point to corresponding objects. Not so with applicant's invention because synchronization is focused on a simple variable existing in an object that represents a shared resource.

The system of Lindholm et al. can be used to synchronize an object using three approaches, but not one of them uses a variable in an object for synchronization. *See* Lindholm et al. at column 3, line 63, to column 4, line 33. The first of these three approaches consists of the use of a synchronized method of an object. When a synchronized method of an object is executed by a thread, the object that contains the synchronized method will be synchronized with and owned by that thread until the execution of the method is terminated. The second approach is the use of critical sections within a method of an object to cause synchronization with a thread when the section protected by a corresponding critical section is executed by the thread. The final approach is the use of a method of an object that contains a call to another object or a method of another object, which is declared to be synchronized. In summary, the system of Lindholm et al. uses specialized methods to synchronize whereas applicant's invention synchronizes objects by the use of simple variables. Methods are more complicated to maintain than variables. Moreover, methods require greater computing resources than variables. No one skilled in the art would choose a more expensive and less efficient technique to synchronize objects. Thus, the technical distinction between method members of an object and a variable member of the object should be kept distinct to understand applicant's invention.

<u>The Office has indicated that it is true that Lindholm et al. lacks a variable that can be used as a pointer to a lock object and/or be used to indicate the lock status</u>. *See* the final Office

Action at page 7, lines 7-9. However, the Office has insisted that it is inherent for objects to have variables and that Lindholm et al. teaches using a variable at column 2, lines 15-23 (referenced as object identifier) to associate an object with a lock, but that the object identifier of Lindholm et al. is stored in the lock. Furthermore, the Office has described various ways to associate two objects, and concluded that "it would have been obvious to apply the teaching of using a variable of the object as a pointer to the lock as taught by Brown [et al.] to the invention of Lindholm [et al.] because there are multiple ways to associate objects."

As indicated by M.P.E.P. Section 2112, the fact that a certain result or characteristic may occur or be present in the prior art is not sufficient to establish the inherency of that result or characteristic (citing favorably *In re Rijckaert*, 9 F. 3d 1531, 1534, 28 U.S.P.Q.2d 1955, 1957 (Fed. Cir. 1993)). Additionally, inherency may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient. *See* M.P.E.P. Section 2112 (citing favorably *In re Robertson*, 169 F. 3d 743, 745, 49 U.S.P.Q.2d 1959, 1950-51 (Fed. Cir. 1999)). Because the Office has admitted that Lindholm et al. lacks a variable that can be used as a pointer to point to a lock object and/or be used to indicate a lock status, it is difficult to understand how such a variable as recited and described by the present invention could be inherent in the disclosure of Lindholm et al. Moreover, M.P.E.P. Section 2143.01 indicates that the mere fact that references can be combined while modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination. *See In re Mills*, 916 F.2d 680, 16 U.S.P.Q.2d 1430 (Fed. Cir. 1990). For argument purposes, applicant generally agrees that there are multiple ways to create an association between two objects. However, one skilled in the art would not be motivated to choose an option that is less efficient to associate an object with a lock object. The motivation "because there a multiple ways to associate objects" is no motivation at all because it does not provide the desirability to combine Lindholm et al. with Brown et al., whose combination applicant respectfully denies.

Like applicant's invention, the system of Brown et al. is directed to provide a locking mechanism, which is bound to an object by allowing the object to contain a pointer to the locking mechanism. But the similarity ends there. To understand the differences between the system of Brown et al. and the system of applicant's invention, it should be noted that the term "locking mechanism" as used by Brown et al. may be likened to a lock object of the applicant's invention but cannot be equated to the recyclable locking mechanism. The main problem is that the locking mechanism of Brown et al. remains bound to the object for the life of the object. *See* the Abstract of Brown et al.

As further emphasized at column 9, lines 52-54, Brown et al. indicates that "once a locking mechanism is bound to an object, it remains assigned to the object for the life of the object." Only when the object is destroyed (such as by garbage collection) will Brown et al. allow another object to use the locking mechanism entry in the locking mechanism table vacated by the destroyed object. In contrast, a lock of applicant's invention returns to the pool of locks whether or not the object that the lock previously secured is destroyed. In other words, a lock object of applicant's invention would return to the pool of locks because a thread no longer needs access to a resource being represented by an object—not because the object was destroyed.

The difficulty with the system of Brown et al. is that unless objects are destroyed, there can be no free locking mechanisms for other threads to use to synchronize access to shared resources. Consider the situation where no object can be destroyed. The system of Brown et al. would permanently stay the execution of multiple threads such that they will have to wait indefinitely to obtain a locking object (which these threads will never in fact obtain) to access shared resources. Not only would this create inadvertent deadlocks, but to solve this problem, Brown et al. would need to provide far more locking objects than can be anticipated. This would be a waste of computing resources.

Clearly, neither Lindholm et al., Brown et al., or Kishimoto, alone much less in combination, teaches or suggests the subject matter of Claim 1. More specifically, none of these references, alone much less in combination, teaches or suggests a recyclable locking mechanism for associating a lock from the pool of locks with an object using the variable of the object as a pointer when requested by a thread, and wherein the lock returns to the pool of locks without having to destroy the object

In the abstract, the system of Lindholm et al., unlike Brown et al., does not assign a synchronization construct to an object for the life of the object: "[F]or each specific thread that seeks de-synchronization with a specific object when a specific synchronization construct . . . is currently located for synchronizing the specific thread with the specific object, <u>the cache manager re-allocates the specific synchronization construct for synchronizing a waiting thread</u> . . . <u>with the specific object.</u> . . ." The system of Brown et al., on the other hand, indicates that "once a locking mechanism is bound to an object, <u>it remains assigned to the object for the life of the object</u>." *See* Brown et al. at column 9, lines 52-54. To combine, either the approach of Brown et al., <u>which binds</u> a locking mechanism to the object for the life of the object, must be abandoned, or the approach of Lindholm et al., <u>which does not bind</u> a synchronizing construct to the object the life of the object, must be jettisoned, and the combination would destroy the operation of either reference. Even if the combination of Lindholm et al. and Brown et al. were possible, which applicant specifically denies, these references cannot anticipate or render applicant's invention obvious.

But there is more. In the Field of the Invention, the system of Lindholm et al. is described as requiring "an object synchronization module and associated method that <u>uses a cache of monitors</u> for synchronizing objects." Brown et al. criticizes the use of a cache of monitors: "While the use of monitors prevents race conditions from occurring, this approach can significantly degrade the performance of the information handling system." *Id.* at column 3, lines 10-12. The Office has insisted that "Lindholm teaches (column 2, lines 24-31) allocating a synchronizing construct for to [*sic*] synchronize an object when no monitor is allocated to synchronize the object." *See* final Office Action dated September 24, 2002, at page 8, lines 4-6. That is not a correct reading of Lindholm et al. At column 2, lines 24-31, Lindholm et al. teaches as follows: "<u>When a respective thread seeks synchronization with a respective object and none of the monitors is allocated for synchronizing the respective object with any of the threads</u> the cache manager removes the respective synchronization construct from the free list, allocates the respective synchronization construct to the respective object, and assigns the respective synchronization construct to synchronize the respective thread with the respective object." The phrase "none of the monitors is allocated for synchronizing the respective object" means that there are free monitors that can be allocated to help synchronize an object with a thread. If there are no free monitors (in other words, all of them have been allocated for synchronizing among other threads) the requesting thread is placed in a waiting list. To combine, either the approach of Lindholm et al., which uses a cache of monitors, must be abandoned or the approach of Brown et al., which criticizes the use of a cache of monitors, must be jettisoned, and the combination would destroy the operation of either reference. Even if the combination of Lindholm et al. and Brown et al. were possible, which applicant again specifically denies, these references cannot anticipate or render applicant's invention obvious.

The Office has sought to combine Kishimoto with Lindholm et al. and Brown et al. As previously discussed, to transfer programs and data among processors, Kishimoto uses 32-bit segment numbers. Kishimoto teaches that the higher 27 bits (from bit 31 to bit 5) indicate an address that stores 32 bits that show the allocation states of 32 segment numbers represented using the lower five bits (from bit 4 to bit 0). In contrast, applicant's claimed invention recites that the set of high bits, which comprises 27 bits, defines the pointer to a lock, and a set of low bits, which comprises five bits, defines a status variable. No lock and no pointer to a lock is taught by Kishimoto. Moreover, no status variable is taught by Kishimoto. The lower 5 bits of Kishimoto indicate segment numbers--but not status of any kind. Additionally, the higher 27 bits of Kishimoto indicate an address that stores 32 bits that show the allocation states of 32 segment numbers--but not a lock or a pointer to a lock of any kind. In construing Claim 5, the Office has completely disregarded the claim limitations of Claim 4 from which Claim 5 depends. As indicated by M.P.E.P. Section 2143.03, to establish *prima facie* obviousness of a claimed invention, all the claim limitations must be taught or suggested by the prior art.

Clearly, neither Lindholm et al., Brown et al., or Kishimoto, alone much less in combination, teaches or suggests the subject matter of Claim 1. More specifically, none of these references, alone much less in combination, teaches or suggests a recyclable locking mechanism for associating the lock from the pool of locks with an object using the variable of the object as a pointer when requested by a thread, returning the lock to the pool of locks without having to destroy the object when the thread no longer needs to access the resource that is represented by the object, as recited in Claim 1.

As will be readily appreciated in the foregoing discussion, none of the three cited and applied references teaches or suggests the subject matter of Claim 1. Specifically, none of the cited and applied references teaches a recyclable locking mechanism in the manner recited in Claim 1. As a result, applicant submits that Claim 1 is clearly allowable in view of the teaching of the references.

With respect to Claims 2-10, all of which depend directly or indirectly from Claim 1, it is clear that the subject matter of these claims is also not taught or suggested by the cited and applied references, namely, Lindholm et al., Brown et al., or Kishimoto. Claims 2-10 all have limitations that are clearly not taught or suggested by any of the cited and applied references, particularly when the limitations are considered in combination with these recitations of the claims from which these claims individually depend. In summary, Claims 2-10 are submitted to be allowable for reasons in addition to the reasons why Claim 1 is submitted to be allowable.

Independent Claim 11 is directed to a method. The method of Claim 11 is recited as comprising asserting an instruction by a thread to lock an object. The method is further recited to include increasing a variable of the object where the variable has a set of high bits for representing a pointer to a lock and a set of low bits for representing a lock status. The method as yet further recites determining whether the variable is greater than a boundary value so as to allocate the lock. And furthermore, the method is recited to include an act of recycling the lock by returning the lock to a pool of locks when the thread no longer needs the object regardless of whether the object persists after the lock returns to the pool of locks. Among other differences, none of the cited and applied references teaches a variable of the object that has a set of high bits for representing a pointer to a lock and a set of low bits for representing a lock status, as recited in Claim 11. Moreover, none of the cited and applied references teaches recycling the lock by returning the lock to a pool of locks when the thread no longer needs the object regardless of whether the object persists after the lock returns to the pool of locks, as further recited in Claim 11. For generally the same reasons discussed above with respect to Claim 1, applicant submits that the subject matter of Claim 11 is not taught or suggested by any of the cited and applied references, and thus, that Claim 11 is also allowable.

With respect to dependent Claims 12-16, all of which depend directly or indirectly from Claim 11, it is also clear that the subject matter of these claims is also not taught or suggested by the cited and applied references, namely, Lindholm et al., Brown et al., or Kishimoto. Claims 12-16 all have limitations that are clearly not taught or suggested by any of the cited and applied references, particularly when the limitations are considered in combination with these recitations of the claims from which these claims individually depend. In summary, Claims 12-16 are submitted to be allowable for reasons in addition to the reasons why Claim 11 is submitted to be allowable.

Independent Claim 17 is directed to a computer. In many ways, the subject matter of independent Claim 17 mirrors the subject matter of the system recited in Claim 1 and the method recited in Claim 11, albeit in a different manner. For reasons generally similar to reasons discussed above with respect to Claims 1-16, Claim 17 is submitted to recite subject matter that is clearly not taught or suggested by any of the cited and applied references. Specifically, none of the cited and applied references teaches or even suggests a recyclable locking mechanism program executed by the processor from the medium to associate a lock with an object using a variable of the object as a pointer when requested by a thread, and the lock is capable of returning to a pool of locks without having to destroy the object when the object is no longer needed by the thread, as recited in Claim 17. As a result, applicant respectfully submits that Claim 17 is allowable. With respect to dependent Claim 18, which depends directly from Claim 17, it is also clear that the subject matter of this claim is also not taught or suggested by the cited and applied references, namely, Lindholm et al., Brown et al., or Kishimoto. Claim 18 adds limitations that are clearly not taught or suggested by any of the cited and applied references, particularly when the limitations are considered in combination with the recitations of Claim 17 from which Claim 18 depends. In summary, Claim 18 is also submitted to be allowable for reasons in addition to the reasons why Claim 17 is submitted to be allowable.

Independent Claim 19 is directed to a computer-readable medium. The computer-readable medium recited in Claim 19 has a recyclable locking mechanism program stored thereon for execution on a computer to associate a lock with an object using a variable of the object as a pointer when requested by the thread. The lock recited by Claim 19 is capable of returning to a pool of locks without having to destroy the object when the object is no longer needed by the thread. For generally the same reasons discussed above with respect to Claims 1, 11, and 17, applicant submits that the subject matter of Claim 19 is not taught or suggested by any of the cited and applied references, and thus, that Claim 19 is also allowable. With respect to Claim 20, which depends directly from Claim 19, it is clear that the subject matter of Claim 20 is also not taught or suggested by the cited and applied references, namely, Lindholm et al., Brown et al., or Kishimoto. Claim 20 adds limitations that are clearly not taught or suggested by any of the cited and applied references, particularly when the limitations are considered in combination with the recitations of Claim 19 from which Claim 20 individually depends. In summary, Claim 20 is submitted to be allowable for reasons in addition to the reasons why Claim 19 is submitted to be allowable.

In light of the foregoing remarks, it is clear that none of the cited and applied references teaches, let alone renders unpatentable, Claims 1-20. The cited and applied references are all directed to a locking technique that works in an entirely different manner from the present invention; requires a locking mechanism to be bound to the life of the object; or has nothing to do with synchronization. The present invention is directed to an entirely different concept and solution. The present application is directed to a recyclable locking mechanism to associate a lock with an object using a variable of the object as a pointer when requested by a thread, and the lock is capable of returning to a pool of locks without having to destroy the object when the object is no longer needed by the thread.

CONCLUSION

In view of the foregoing remarks, applicant submits that all of the claims in the present application are clearly patentably distinguishable over the teachings of Lindholm et al., Brown et al., and Kishimoto. Thus, applicant submits that this application is in condition for allowance. Reconsideration and reexamination of the application and allowance of the claims and passing of the application to issue at an early date are solicited. If the Examiner has any remaining questions concerning this application the Examiner is invited to contact the applicant's undersigned attorney at the number below.

Respectfully submitted,

CHRISTENSEN O'CONNOR
JOHNSON KINDNESS^PLLC

[illegible]

D.C. Peter Chu
Registration No. 41,676
Direct Dial No. 206.695.1636

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the U.S. Postal Service in a sealed envelope as first class mail with postage thereon fully prepaid and addressed to the U.S. Patent and Trademark Office, P.O. Box 2327, Arlington, VA 22202, on the below date.

Date: November 25, 2002 Cindy A. Norton

DPC:clm

LAW OFFICES OF
CHRISTENSEN O'CONNOR JOHNSON KINDNESS^PLLC
1420 Fifth Avenue, Suite 2800
Seattle, Washington 98101
206.682.8100